Information Processing Letters 108 (2008) 160–164

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

An impossibility about failure detectors in the iterated immediate snapshot model

Sergio Rajsbaum [a], Michel Raynal [b,*], Corentin Travers [b]

[a] Instituto de Matemáticas, Universidad Nacional Autónoma de México, D.F. 04510, Mexico
[b] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

ARTICLE INFO

Article history:
Received 27 November 2007
Received in revised form 18 April 2008
Available online 9 May 2008
Communicated by C. Scheideler

Keywords:
Asynchronous shared memory system
Atomic read/write register
Distributed computing
Distributed computability
Failure detector
Iterated immediate snapshot model
Process crash
Snapshot operation

ABSTRACT

The *Iterated Immediate Snapshot* model (IIS) is an asynchronous computation model where processes communicate through a sequence of one-shot *Immediate Snapshot* (IS) objects. It is known that this model is equivalent to the usual asynchronous read/write shared memory model, for wait-free task solvability. Its interest lies in the fact that its runs are more structured and easier to analyze than the runs in the shared memory model. As the IIS model and the shared memory model are equivalent for wait-free task solvability, a natural question is the following: Are they still equivalent for wait-free task solvability, when they are enriched with the same failure detector? The paper shows that the answer to this question is "no".

1. Introduction

The *Iterated Immediate Snapshot* model (IIS) of Borowsky and Gafni [5] is an asynchronous computation model where the processes communicate through a sequence of one-shot *Immediate Snapshot* (IS) objects. Each IS object can be accessed with a single operation denoted write_snapshot(), that atomically writes a value and returns a snapshot of its contents. Each process can access each IS object at most once. Processes access the sequence of IS objects, one-by-one, in the same order, and asynchronously; moreover, any number of processes can crash. It has been shown by Borowsky and Gafni that this model is equivalent to the usual read/write shared memory model, for wait-free task solvability. Its interest lies in the fact that its runs are more structured and easier to analyze than the runs in the shared memory model. As the IIS model and the shared memory model are equivalent for wait-free task solvability, a natural question is the following: Are they still equivalent for wait-free task solvability, when they are enriched with the same failure detector? The paper shows that the answer to this question is "no". Finally, the paper discusses alternative ways of studying failure detectors within the IIS framework.

2. The iterated immediate snapshot (IIS) model

One-shot immediate snapshot. A one-shot immediate snapshot object *IS* abstracts a shared array $SM[1..n]$ with one entry per process. That array is initialized to $[\bot, \ldots, \bot]$, where $\bot$ is a default value that cannot be written by a process. Intuitively, when a process p_i invokes IS.write_snapshot(v), it is as if it instantaneously executes a $SM[i] \leftarrow v$ operation followed by a snapshot [1,3] of the whole shared array. If several processes execute IS.write_snapshot() simultaneously, then their corresponding write operations are executed concurrently, followed by a concurrent execution of their snapshot operations.

* Corresponding author.
E-mail addresses: rajsbaum@math.unam.mx (S. Rajsbaum), raynal@irisa.fr (M. Raynal), ctravers@irisa.fr (C. Travers).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2008.05.001

For each p_i, the write_snapshot() operation satisfies the three following properties, where v_i is the value written by p_i and sm_i is the *view* it gets back from the operation. We consider sm_i as a set of pairs (k, v_k), where v_k corresponds to the value in p_k's entry of the array. If $SM[k] = \perp$, the pair $(k, \perp)$ is not placed in sm_i. Also, we define $sm_i = \emptyset$, if the process p_i never invokes write_snapshot() on the corresponding object. These properties are:

- Self-inclusion. $\forall i$: $(i, v_i) \in sm_i$.
- Containment. $\forall i, j$: $sm_i \subseteq sm_j \vee sm_j \subseteq sm_i$.
- Immediacy. $\forall i, j$: $(i, v_i) \in sm_j \Rightarrow sm_i \subseteq sm_j$.

The first property holds because a process sees the value it has written. The second property states that the *views* (i.e., the contents of the sm_i sets) obtained by the processes can be ordered by containment. The last property states that when a process invokes write_snapshot(), the snapshot is scheduled immediately after the write.[1]

The write_snapshot() operation can be wait-free implemented is the classical single-writer/multi-reader atomic registers model [4] (1WMR) (for completeness, this implementation is described in Appendix A). The set of the write_snapshot() invocations is *set-linearizable* [21]. This means that each write_snapshot() issued by a process appears as if it has been instantaneously executed at a single point of the time line, without preventing several write_snapshot() to appear at the same point of time.

The iterated immediate snapshot model (IIS). In the IIS model the shared memory is made up of an infinite number of one-shot immediate snapshot objects $IS[1], IS[2], \ldots$ These objects are accessed sequentially (and asynchronously) by the processes according to the following round-based pattern:

$r_i \leftarrow 0$; $v_i \leftarrow$ local_input;
loop forever $r_i \leftarrow r_i + 1$;
$sm_i \leftarrow IS[r_i]$.write_snapshot(v_i);
local computation; (* perhaps updating v_i *)
end loop.

3. Why the IIS model is important

The interest of the IIS model comes from its seemingly restrictive, round-by-round nature. It restricts the set of interleavings of the shared memory model without restricting the power of the model. Its runs have an elegant recursive structure: the structure of global states after $r+1$ rounds is easily obtained from the structure of the global states after r rounds. This implies a strong correlation with topology, and allows for an easier analysis of wait-free asynchronous computations. Indeed, the IIS model was the basis for the proof in [5] of the main characterization theorem of [18], and was instrumental for the results of [13] and of [22].

[1] The immediacy property can be rewritten as $\forall i, j$: $((i, v_i) \in sm_j \wedge (j, v_j) \in sm_i) \Rightarrow sm_i = sm_j$. Thus, concurrent invocations of write_snapshot() obtain the same view.

3.1. Decision tasks

Definition. A *decision task* is a one-shot decision problem specified in terms of an input/output relation Δ. The processes start with private input values, and must eventually decide on output values, by writing to a write-once register. An *input vector* I specifies in its ith entry, $I[i]$, the input value of process p_i, and we say p_i *proposes* $I[i]$ in the execution. Similarly, an *output vector* J specifies a decision value $J[i]$ for each process p_i. A task defines a set of legal input vectors, and for each one, Δ specifies a set of legal output vectors. Thus, given input vector I, the processes decide a vector J such that (1) $J \in \Delta(I)$, and (2) individually each p_i decides $J[i]$ or crashes. It is sometimes convenient to consider *inputless tasks,* where a process has only one possible input value, namely its own id. (For the interested reader, a formal definition of a task is given in Section 2.1 of [18].) A *bounded* decision task is a task whose number of input vectors is finite.

Examples of tasks. The most famous decision task is the *consensus* problem [10]. Each process proposes a value and the correct processes have to decide the same value, that has to be a proposed value. So here, an output vector contains the same value in all entries. The relation Δ states that the single value present in an output vector is a value that appears in the corresponding input vector [20].

In the *k-set agreement* problem up to k different values can be decided [8]. Other examples are the *committee decision* problem [2,14], and the *musical benches* problem [12].

3.2. A fundamental result in IIS the model

Let us observe that the IIS model requires each correct process to execute an infinite number of rounds. However, it is possible that a correct process p_1 is unable to receive information from another correct process p_2. Consider a run where both execute an infinite number of rounds, but p_1 is scheduled before p_2 in every round. Thus, p_1 never reads a value written to an immediate snapshot object by p_2. Of course, in the usual (non-iterated read/write shared memory) asynchronous model, two correct processes can always eventually communicate with each other. Thus, it may be surprising that, despite the use of such a strong constraint on the behavior of the processes, it is still possible to derive simulations between the IIS model and the base read/write non-iterated model, as stated in the following theorem.

Theorem 1. *(See Borowsky and Gafni, 1997 [5].) A bounded decision task can be wait-free solved in the 1WMR register model if and only if it can be wait-free solved in the IIS model.*

4. The read/write model enriched with failure detectors

4.1. The wait-free 1WMR shared memory model

This model consists of n processes, $p_1, \ldots, p_n$, that communicate through a shared memory. A process behaves correctly until it possibly crashes. A process that does not crash in a run is *correct* in that run, otherwise

it is faulty. *Wait-free* means that any number of processes can crash [17].

The shared memory is made up of 1WMR registers, and structured as an array $SM[1..n]$, such that only p_i can write to $SM[i]$, and any process can read any entry. A process can have local variables (those are denoted with sub-indexed lowercase letters, e.g., $local_i$).

4.2. Failure detectors

The concept of a *failure detector* has been introduced by Chandra and Toueg [6] for the message passing model. Since then it has been widely studied, also in the shared memory model (e.g., [15,19]). Informally, a failure detector is a device that provides each process p_i with information about process failures, through a local variable FD_i that p_i can only read. Several classes of failure detectors can be defined according to the kind and the quality of the information on failures that has to be delivered to the processes.

An example. Consider the class denoted $\diamond\mathcal{S}$, defined in [6]. A failure detector of that class provides each process p_i with a local variable SUSPECTED_i that contains identities of processes that are believed to have crashed. When $j \in \text{SUSPECTED}_i$ we say "p_i suspects p_j". The failure detector class $\diamond\mathcal{S}$ is defined by the following properties:

- Strong completeness. There is a time after which every faulty process is permanently suspected by every correct process.
- Eventual weak accuracy. There is a time τ, after which there is a correct process p_ℓ that is never suspected by all the correct processes.

The time τ, and the process p_ℓ are not explicitly known. These properties do not prevent an initial arbitrarily long period during which the sets SUSPECTED_i contain arbitrary values; they only state that this anarchy period eventually terminates. Also, notice that a failure detector of the class $\diamond\mathcal{S}$ can make an infinite number of mistakes (e.g., looping on suspecting and not suspecting correct processes).

The class $\diamond\mathcal{S}$ is particularly important from an asynchronous computability point of view. Namely, it is the weakest class of failure detectors that allows solving the consensus problem despite asynchrony and process crashes [7]; any other class of failure detector that allows solving consensus despite asynchrony and crashes provides information on failures that includes the information provided by $\diamond\mathcal{S}$. Examples of other failure detector classes are described in [6,9,23].

5. An impossibility result

This section considers the question posed in the introduction: are the basic read/write shared memory model and the IIS model equivalent for wait-free decision task solvability, when both are equipped with a failure detector of the same class? As announced, this section shows that the answer to that question is "no".

(1) **init** $r_i \leftarrow 0$; $sm_i^{(0)} \leftarrow \{\langle input_i, \emptyset, dec_i\rangle\}$; $dec_i \leftarrow g(sm_i^{(0)})$;
(2) **loop forever**
(3) $\quad r_i \leftarrow r_i + 1$;
(4) $\quad local_i \leftarrow compute(sm_i^{(r_i-1)}, \text{FD}_i)$;
(5) $\quad sm_i^{(r_i)} \leftarrow IS[r_i].\text{write\_snapshot}(\langle sm_i^{(r_i-1)}, local_i, dec_i\rangle)$;
(6) $\quad$ **if** $(dec_i = \perp)$ **then** $dec_i \leftarrow g(sm_i^{(r_i)})$ **end if**
(7) **end loop**.

Fig. 1. Full information "IIS + Failure detector" code for p_i.

5.1. An IIS model with failure detector

Assume a failure detector of some class C, is available in the IIS model, that provides each process p_i with a local variable FD_i. During each round r, p_i can read FD_i any number of times, and eventually, access the next immediate snapshot object. Also, assume some task T is being solved, so that each process starts with a private input value in a local variable $input_i$, and must eventually put its decision in a local write-once variable dec_i, initially $\perp$.

The round-based framework of the IIS model defined in Section 2 is refined as described in Fig. 1, with a *full information* algorithm $\mathcal{A}$ solving the task T. In line (4), $compute(sm_i^{(r_i-1)}, \text{FD}_i)$ is a shorthand for a loop that is executed by p_i, where in each iteration it considers the current value of the failure detector obtained from FD_i, to make local computations and decide weather to execute one more iteration or to exit the loop returning a value to be placed in the variable $local_i$. It is assumed that the number of iterations executed is finite. When line (5) is executed, p_i invokes $IS[r]$.write_snapshot() to write its view $sm_i^{(r-1)}$ (obtained from the previous write_snapshot() invocation) together with its latest failure detector information (or any additional desired information), and the current decision dec_i. After it has obtained a view $sm_i^{(r)}$ during the round r (line 5), a process p_i checks if it can decide by applying a decision function, denoted $g()$, to that view $sm_i^{(r)}$. Recall that each correct process keeps on taking steps forever, even after having decided.

We say that *a task T is wait-free solvable in the IIS model with C* if there is an algorithm $\mathcal{A}$ of the form in Fig. 1, such that for any failure detector of the class C, in any (infinite) run where the input values are in the domain of T, every correct process eventually decides, and the decisions satisfy the input/output relation Δ that defines the task T.

5.2. The impossibility

The idea of the proof is to show that C does not restrict the set of possible interleavings of the IIS model. Thus, if T is solvable in the IIS model with C, in particular it is solvable in the set of runs of the IIS model, and hence solvable in the read/write shared memory model, by Theorem 1. The crucial step is to group together all operations of a round related to the failure detector, in a fixed predetermined order, before executing shared memory operations of that round. And only then, considering all interleavings of the shared memory operations.

Theorem 2. *For any failure detector class C and task T, if T is solvable in the IIS model with C then T is wait-free solvable in*

the base read/write shared memory model with no failure detector.

Proof. To facilitate the proof, we consider a single input configuration of T (e.g., [5]), and hence a single input initial configuration of the system, denoted S^0.

We consider the following subset of runs of the IIS model with C, where no process fails, defined inductively, starting with S^0. Consider some reachable configuration of the system after $r-1$ rounds, say S^{r-1} (for the basis, we take S^0). We schedule the steps of the processes following the same round structure of the algorithm $\mathcal{A}$, by having all processes execute their round r before proceeding to round $r+1$. Moreover, we schedule first all local computations of the processes corresponding to line (4) of round r, before any process starts executing its line (5). We schedule all those local operations in a fixed order, first all those of p_1, then all those of p_2, until all those of p_n, and we get a specific partial run

$$compute\big(sm_1^{(r-1)}, \text{FD}_1\big), compute\big(sm_2^{(r-1)}, \text{FD}_2\big),$$
$$\ldots, compute\big(sm_n^{(r-1)}, \text{FD}_n\big).$$

Let us denote the system configuration at the end of this partial run by S_1^{r-1}. Now, we consider all possible interleavings of executions of line (5) for all processes. After such an interleaving, we execute (in an arbitrary order) line (6) of every process, and end up in a configuration denoted S^r (with a slight abuse of notation, as for each such interleaving the system ends up in a different configuration).

Let us observe that any failure detector output change at a process p_i, after p_i returned from its invocation $compute(sm_i^{(r-1)}, \text{FD}_i)$ does not affect the execution of its operation of line (5), because the value to be written by the write_snapshot() invocation is fixed. That is, the views obtained as a result of such invocations, in the $sm_i^{(r)}$ variables, on all possible interleavings, are equivalent to the views of the IIS model with no failure detector. In other words, given two set-linearizations of the write_snapshot() operations, the view of a process p_i is the same in both, iff in the IIS model with no failure detector, p_i has the same view in both set-linearizations.[2]

We have constructed a subset of runs of IIS with C where the views of the processes at the end of each round have the same structure as the views of the original IIS model with no failure detector. As we are assuming that the algorithm $\mathcal{A}$ solves T in the IIS with C, it solves T also in the IIS model alone. As the IIS model and the read/write model are computationally equivalent for wait-free task solvability (Theorem 1), the result follows. □

Remark. It follows from the previous theorem that, to wait-free solve a decision task, failure detectors are useless in the IIS model. (From a practical point of view, this means that à la Paxos consensus algorithms (e.g., [11,16]) cannot be devised for the IIS model.) However and interestingly, it appears that there are failure detector classes C such that adding appropriate restrictions on the IIS model (i.e., restricting its set of runs) provides a computation model that is wait-free equivalent to the read/write model enriched with a failure detector of the corresponding class C. This approach has given rise to the IRIS (Iterated Restricted Immediate Snapshot) model that is proposed and investigated in [22].

[2] In the topology parlance, this can be formulated as follows: the complex of views in both models are isomorphic.

```
operation write_snapshot(v_i):
    REG[i] ← v_i;
    repeat LEVEL[i] ← LEVEL[i] − 1;
           for j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end for;
           view_i ← {j: level_i[j] ⩽ LEVEL[i]};
    until (|view_i| ⩾ LEVEL[i]) end repeat;
    return({(j, REG[j]) such that j ∈ view_i}).
```

Fig. 2. Borowsky–Gafni's one-shot write_snapshot() algorithm (code for p_i).

Appendix A. A wait-free implementation of the write_snapshot() operation

For a completeness purpose, this appendix presents a one-shot write_snapshot() construction. This algorithm, due to Borowsky and Gafni [4], is described in Fig. 2. That algorithm considers a one-shot immediate snapshot object (a process invokes IS.write_snapshot() at most once). It uses two arrays of 1W*R atomic registers denoted $REG[1..n]$ and $LEVEL[1..n]$ (only p_i can write $REG[i]$ and $LEVEL[i]$). A process p_i first writes its value in $REG[i]$. Then the core of the implementation of write_snapshot() is based on the array $LEVEL[1..n]$. That array, initialized to $[n+1, \ldots, n+1]$, can be thought of as a ladder, where initially a process is at the top of the ladder, namely, at level $n+1$. Then it descends the ladder, one step after the other, according to predefined rules until it stops at some level (or crashes). While descending the ladder, a process p_i registers its current position in the ladder in the atomic register $LEVEL[i]$.

After it has stepped down from one ladder level to the next one, a process p_i computes a local view (denoted $view_i$) of the progress of the other processes in their descent of the ladder. That view contains the processes p_j seen by p_i at the same or a lower ladder level (i.e., such that $level_i[j] \leqslant LEVEL[i]$). Then, if the current level ℓ of p_i is such that p_i sees at least ℓ processes in its view (i.e., processes that are at its level or a lower level) it stops at the level ℓ of the ladder. Finally, p_i returns a set of pairs determined from the values of $view_i$. Each pair is a process index and the value written by the corresponding process. This behavior is described in Fig. 2 [4].

This very elegant algorithm satisfies the following properties [4]. The sets $view_i$ of the processes that terminate the algorithm, satisfy the following main property: if $|view_i| = \ell$, then p_i stopped at the level ℓ, and there are ℓ processes whose current level is $\leqslant \ell$. From this property, follow the self-inclusion, containment and immediacy properties (stated in Section 2) that define the one-shot immediate snapshot object.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, Journal of the ACM 40 (4) (1993) 873–890.

[2] Y. Afek, E. Gafni, S. Rajsbaum, M. Raynal, C. Travers, Simultaneous consensus tasks: A tighter characterization of set consensus, in: Proc. 8th Internat. Conference on Distributed Computing and Networking (ICDCN'06), in: LNCS, vol. 4308, Springer-Verlag, 2006, pp. 331–341.

[3] H. Attiya, O. Rachman, Atomic snapshots in $O(n \log n)$ operations, SIAM Journal on Computing 27 (2) (1998) 319–340.

[4] E. Borowsky, E. Gafni, Immediate atomic snapshots and fast renaming, in: Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93), 1993, pp. 41–51.

[5] E. Borowsky, E. Gafni, A simple algorithmically reasoned characterization of wait-free computations, in: Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97), ACM Press, 1997, pp. 189–198.

[6] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.

[7] T. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, Journal of the ACM 43 (4) (1996) 685–722.

[8] S. Chaudhuri, More choices allow more faults: Set consensus problems in totally asynchronous systems, Information and Computation 105 (1993) 132–158.

[9] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, S. Toueg, The weakest failure detectors to solve certain fundamental problems in distributed computing, in: Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04), ACM Press, 2004, pp. 338–346.

[10] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32 (2) (1985) 374–382.

[11] E. Gafni, L. Lamport, Disk Paxos, Distributed Computing 16 (1) (2003) 1–20.

[12] E. Gafni, S. Rajsbaum, Musical benches, in: Proc. 19th Internat. Symposium on Distributed Computing (DISC'05), in: LNCS, vol. 3724, Springer-Verlag, 2005, pp. 63–77.

[13] E. Gafni, S. Rajsbaum, M. Herlihy, Subconsensus tasks: Renaming is weaker than set agreement, in: Proc. 20th Internat. Symposium on Distributed Computing (DISC'06), in: LNCS, vol. 4167, Springer-Verlag, 2006, pp. 329–338.

[14] E. Gafni, S. Rajsbaum, M. Raynal, C. Travers, The committee decision problem, in: Proc. Latin American Theoretical Informatics Symposium (LATIN'06), in: LNCS, vol. 3887, Springer-Verlag, 2006, pp. 502–514.

[15] R. Guerraoui, P. Kouznetzov, Failure detectors as type boosters, Distributed Computing 20 (5) (2008) 343–358.

[16] R. Guerraoui, M. Raynal, The alpha of indulgent consensus, The Computer Journal 50 (1) (2007) 53–67.

[17] M.P. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 124–149.

[18] M.P. Herlihy, N. Shavit, The topological structure of asynchronous computability, Journal of the ACM 46 (6) (1999) 858–923.

[19] W.-K. Lo, V. Hadzilacos, Using failure detectors to solve consensus in asynchronous shared-memory systems, in: Proc. of the 8th Internat. Workshop on Distributed Algorithms (WDAG'94), in: LNCS, vol. 857, Springer-Verlag, 1994, pp. 280–295.

[20] A. Mostefaoui, S. Rajsbaum, M. Raynal, Conditions on input vectors for consensus solvability in asynchronous distributed systems, Journal of the ACM 50 (6) (2003) 922–954.

[21] G. Neiger, Set Linearizability, Brief announcement, in: Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94), ACM Press, 1994, p. 396.

[22] S. Rajsbaum, M. Raynal, C. Travers, The iterated restricted immediate snapshot model, in: Proc., 14th Annual Internat. Conference on Computing and Combinatorics (COCOON'08), in: LNCS, vol. 5092, Springer-Verlag, Dalian (China), June 2008, pp. 487–496.

[23] M. Raynal, A short introduction to failure detectors for asynchronous distributed systems, ACM SIGACT News, Distributed Computing Column 36 (1) (2005) 53–70.